Automata-Based Refinement Checking for Real-Time Systems

Christian Brenner, Christian Heinzemann,
Wilhelm Schäfer
Software Engineering Group
Heinz Nixdorf Institute, University of Paderborn
Zukunftsmeile 1
33102 Paderborn, Germany
[cbr|c.heinzemann|wilhelm]@uni-paderborn.de

Stefan Henkler
OFFIS
Escherweg 2
26121 Oldenburg, Germany
stefan.henkler@offis.de

Abstract: Model-driven development of real-time safety-critical systems requires to support refinement of behavioral model specifications using, for example, timed simulation or timed bisimulation. Such refinements, if defined properly, guarantee that (safety and liveness) properties, which have been verified for an abstract model, still hold for the refined model. In this paper, we propose an automatic selection algorithm selecting the most suitable refinement definition concerning the type of model specification applied and the properties to be verified. By extending the idea of test automata construction for refinement checking, our approach also guarantees that a refined model is constructed correctly concerning the selected and applied refinement definition. We illustrate the application of our approach by an example of an advanced railway transportation system.

1 Introduction

Innovation in embedded real-time systems is increasingly driven by software [SW07]. Since embedded real-time systems often operate in safety-critical environments, errors in the software may cause severe damages. Thus, ensuring correct operation and safety of the software is mandatory, but challenging due to its high complexity. High complexity is not only a result of the complexity of the system in terms of its size but in addition, due to its strict real-time requirements. Embedded real-time systems require the system (and all its components) to produce the expected (correct) output no later than at a given point in time.

Model-driven software development addresses these challenges by building formal models of the software instead of implementing it directly. These models can be used to verify safety and liveness properties of the system under development using model checking [BK08]. For embedded real-time systems, timed automata [AD94, BY03] have proven to be a suitable model to support model checking [ACD93, BY03]. Model checking, however, does not scale for large systems. Therefore, an underlying component model is defined in such a way that it supports compositional verification, i.e. parts of the model are verifiable independently.

In a little more detail, components, e.g. RailCab and TrackSection in Figure 1, communicate via protocols (specified by timed automata) which define a sequence of timed message exchanges. Message exchange, in turn, is using connectors and, in case of asynchronous communication, buffers for storing incoming messages. Connector and buffer behavior is also specified using timed automata. A so-called abstract model of such a protocol that includes a model of the connector and possibly message buffers is verified to prove that it fulfills a given (safety) property φ. Then, the abstract protocol behavior is assigned to a component and usually refined according to the needs and context of the individual component. Note, that abstract protocols are defined in such a way that they become reusable in various contexts and for various components, possibly even in different systems.

A common approach is to check such a refinement for correctness rather than verifying φ for the refined protocol behavior again. Checking refinement for correctness is guaranteeing a correct refinement according to the definitions as given in Section 3. Such an approach makes formal verification of distributed systems, whose subcomponents communicate via protocols, a lot more scalable. Buffer and connector behavior specifications (e.g. by timed automata) do not need to be taken into account anymore when the refinement of protocols is checked for correctness.

However, a number of different refinement definitions have been proposed in the literature. Depending on the particular type of protocol which is refined, they might all be useful when building a system. In general, a refinement definition needs to be as weak as possible for enabling reuse of an abstract protocol in as many different contexts as possible, but as strong as necessary for guaranteeing that φ holds for the refined protocol behavior. That is especially useful if the same abstract behavior is used multiple times in the same system.

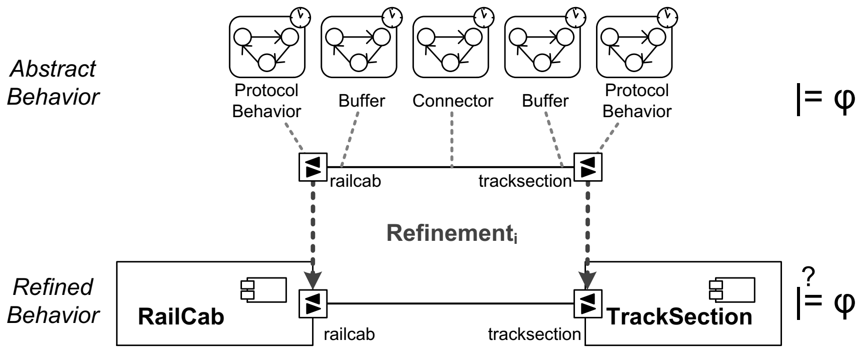

Figure 1: Overview of the Refinement Approach

Currently, the selection of a suitable refinement definition is left to the developer and his expertise without giving him any further tool support. If the developer selects a too weak refinement definition, it is not guaranteed that φ holds for the refined protocol behavior. If the selected refinement definition is too strong, the refinement check might reject the refined protocol behavior although it fulfills φ. This may happen, e.g., if the refined model removes behavior that is irrelevant for φ, but which is checked by the too strong refinement definition. In this paper, we provide an automatic selection algorithm selecting a suitable refinement definition based on the type of model, for example timed vs. untimed, as well as the specification of the property φ. As a basis, we identify the commonalities and differences of the six most relevant existing refinement definitions for distributed real-time systems.

The main contribution of this paper is an algorithm to support automatic checking of a correct protocol refinement based on an extension of the approach described in [JLS00]. In [JLS00], a so-called test automaton is automatically constructed to verify correct refinements. The test automaton encodes both, the abstract model and the constraints of the selected refinement. The constraints specify the allowed deviation of the refined system model from the abstract model. If (and only if) the refined system model violates one of the constraints, the test automaton enters a special error location indicating that the refinement is not correct. However, that approach is restricted to just one refinement definition (namely timed ready simulation which is explained below). Our extensions of [JLS00] provide for the construction of test automata for all the mentioned different refinement definitions. They include, in particular, the notion of asynchronous communication via buffers and thus a very important type of communication and corresponding refinement definition for distributed real-time systems. Checking refinement definitions for that case has not been considered before.

In this paper, we will use the RailCab system[1] as a case study for an embedded real-time system. The vision of the RailCab project is a railway transportation system where autonomous vehicles, called RailCabs, travel on existing track systems. Since RailCabs operate autonomously, collision avoidance on track has to be realized by software, only. For avoiding collisions, each RailCab must register at track sections for gaining admission before entering. This communication is safety-critical and must obey real-time requirements to ensure that a RailCab comes to a stop before entering a track if it has no admission. In our case study, we show how the same abstract behavior can be refined for four different types of track sections. Each type of track section requires the abstract behavior to be refined differently. Using our approach, we succeeded in showing the correctness of the refinements by using different refinement definitions for the different types of track sections.

The paper is structured as follows: In Section 2, we introduce timed automata. Section 3 presents the most relevant refinement definitions and the corresponding selection criteria using the RailCab example. The construction of the test automaton is given in Section 4. We discuss related work in Section 5 before concluding the paper in Section 6.

2 Timed Automata

In our approach, we use timed automata as a behavior model for the components within a system. They extend finite automata by a set of real-valued clocks [AD94, BY03]. Clocks measure the progress of time in a system and allow for the specification of time-dependent

[1]http://www.railcab.de

behavior. In essence, that means that the output of the automaton does not only depend on its inputs, but also on the points in time at which the inputs are received.

Based on its clocks, a timed automaton specifies time guards, clock resets, and invariants. A time guard is a clock constraint that restricts the execution of a transition to a specific time interval. A clock reset sets the value of a clock back to zero while a transition is fired. Invariants are clock constraints associated with locations that forbid that a timed automaton stays in a location when the clock values exceed the value of the invariant. In combination, time guards and invariants define the time intervals where transitions may fire at run-time.

In addition to guards and resets, transitions may carry messages that specify inputs and outputs of the timed automaton. Input messages are denoted by ?, output messages by !. We assume an asynchronous communication of the timed automata. Messages are sent over a connector and put into a buffer on the receiver side as shown in Figure 1. By providing explicit timed automata for the buffers and the connector, we map the asynchronous communication of our timed automata to the synchronous communication of timed automata as used in [BY03].

In timed automata, transitions are not forced to fire instantaneously if they are enabled. Instead, the automaton may rest in a location and delay. For many applications, this is not sufficient because they require that transitions fire immediately if a certain message has been received. As a result, transitions can be marked as *urgent*. If an urgent transition is enabled, it fires immediately without any delay [BGK+96].

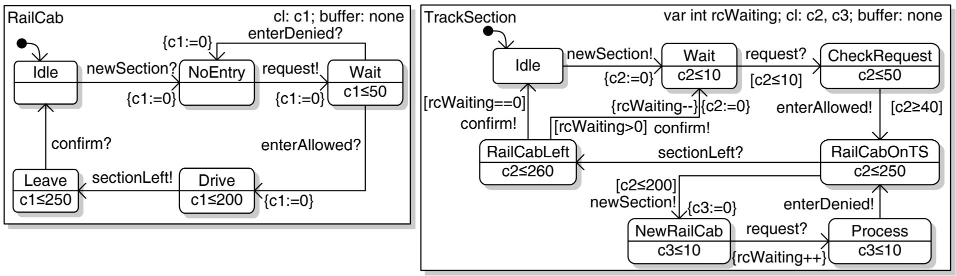

Figure 2: Abstract Behavior for Entering a Track Section

Figure 2 shows an example of two timed automata. They specify the protocol behavior of the abstract model of Figure 1. The automata specify a simplified registration protocol where RailCabs register at a track section to be allowed to enter it. Initially, both automata are in the Idle locations. Then, the track section sends newSection to an approaching RailCab. The RailCab requests to enter the track section by sending a request. The request is received by the track section which sends enterAllowed. Then, the RailCab switches to *Drive* and enters the track section. If another RailCab approaches, the track section sends newSection and switches to NewRailCab. In this case, it denies the request by sending enterDenied. The track section uses the variable rcWaiting to store the number of RailCabs waiting for entry. A RailCab finally sends sectionLeft after it left the track section which is confirmed by the track section. If there are RailCabs waiting, the track section switches to Wait for processing the next request. Otherwise, it switches to Idle.

The interaction of RailCabs and a track section is safety-critical, because RailCabs may come into collision if a RailCab enters a track section after the track section denied the entry. However, we also want to ensure that RailCabs are actually allowed to enter the track section. We use model checking to prove that the model fulfills such safety and liveness properties which ensure correct behavior. In the example, we need to verify two properties. First, we verify "If a track section sends enterDenied, then the RailCab will not send sectionLeft until the track section sends enterAllowed". If sectionLeft occurred before, this would imply that the RailCab entered the track section without being allowed to do so. Second, we verify "In all system states, there exists a path where the track section eventually sends enterAllowed." for checking that progress is possible.

We specify such properties formally by using the timed computation tree logic (TCTL, [ACD93]) and verify them using model checking. In TCTL, the first property is formalized as follows:

$$AG(enterDenied \Rightarrow A(\neg sectionLeft\ W\ enterAllowed)) \tag{1}$$

AG denotes that the formula in parentheses holds globally in all states of all execution paths. An occurrence of enterDenied implies that on all execution paths sectionLeft does not occur ($\neg sectionLeft$) until enterAllowed occurs which is modeled by AW. The property uses a so-called weak until (W) [BK08, pg. 327]. In contrast to the normal until (U) it does not require enterAllowed to occur eventually. A weak until, however, can be mapped to the standard TCTL operators [BK08, pg. 327].

The second property is formalized by:

$$AG(EF\ enterAllowed) \tag{2}$$

The operator EF denotes that enterAllowed is eventually sent.

3 Refinement Definition and Selection

A refinement definition relates an abstract model and a refined model of the same system as shown in Figure 1. It defines how the behavior defined by the refined model may deviate from the behavior defined by the abstract model. A restrictive refinement definition guarantees that verified safety and liveness properties still hold for the refined model. A less restrictive refinement definition leaves developers more flexibility to adapt the abstract model to a component and, thus, allows for more possible refined models. Finding a suitable refinement definition is, thus, a trade-off between flexibility upon building the refined model and properties that are preserved by the refined model.

In this section, we explain the six most relevant refinement definitions for embedded real-time systems informally due to space restrictions. Those are simulation [BK08], bisimulation [BK08], timed simulation [WL97], timed bisimulation [WL97], timed ready simulation [JLS00], and relaxed timed bisimulation [HH11]. For the informed reader, please note that we only consider so-called *weak* variants of the refinements [WL97]. These weak refinements abstract from any internal behavior which is defined by transitions not carrying a message, but performing an internal computation. It is sufficient to consider only weak variants because the protocol specifications, which are the subject of this paper, only specify message exchange between components.

Simulation requires that the refined model only includes sequences of messages that are specified already by the abstract model. The refined model, however, may remove sequences of messages. Thus, simulation preserves any CTL*-formulas [BK08] only containing $\forall$-path quantifiers. Formulas with an $\exists$-path quantifier are not preserved because the path fulfilling the property might be removed. For preserving CTL*-formulas with $\exists$-path quantifiers, we use bisimulation. It requires that the refined model includes exactly the same sequences of messages as the abstract model.

For timed automata, variants of simulation and bisimulation have been developed that impose conditions on the timing of messages. These conditions are absolutely necessary to refine protocols of real-time systems. Like the (untimed) simulation, variants of a timed simulation only preserve properties containing $\forall$-path quantifiers while variants of a timed bisimulation also preserve properties containing $\exists$-path quantifiers.

Timed simulation [WL97] requires that the refined model only includes sequences of messages that are specified already by the abstract model. In addition, the refined model only specifies sending or receiving a message in the same or a restricted time interval. If the abstract model uses urgent transitions, timed simulation is not sufficient. As shown in [JLS00], timed simulation does not guarantee that if the refined model R simulates an abstract model A, the composition with any other model B, $R \parallel B$, simulates $A \parallel B$. As a solution, [JLS00] presents a new refinement definition: the *timed ready simulation*. In addition to the conditions of a timed simulation, it requires the refined model to preserve all urgent transitions including their timing.

A timed bisimulation requires that the refined model includes exactly the same sequences of messages and specifies exactly the same time intervals as the abstract model. Therefore, timed bisimulation is a very strong refinement definition and preserves all TCTL properties. Using an input buffer for messages allows to relax the conditions of timed bisimulation. We call this relaxation *relaxed timed bisimulation* [HH11]. The relaxed timed bisimulation enables to extend the time intervals for received messages, but requires that the upper bounds of time intervals for sending messages are exactly the same as in the abstract model. It preserves all CTL*-formulas and all TCTL formulas only referring to the latest sending of messages.

For illustrating the selection of a refinement definition, we provide examples of two refinements of the abstract track section behavior of Figure 2. Figure 3 shows the behavior of a railroad crossing on the left and the behavior of a normal track section on the right. In addition, the RailCab system contains switches and stations which also execute refined versions of the abstract track section behavior of Figure 2. We omit the behavior descriptions for switches and stations due to space restrictions.

Informally speaking, the two automata specify the following behavior: If a RailCab wants to enter a railroad crossing, the railroad crossing must close the gates. The transition from

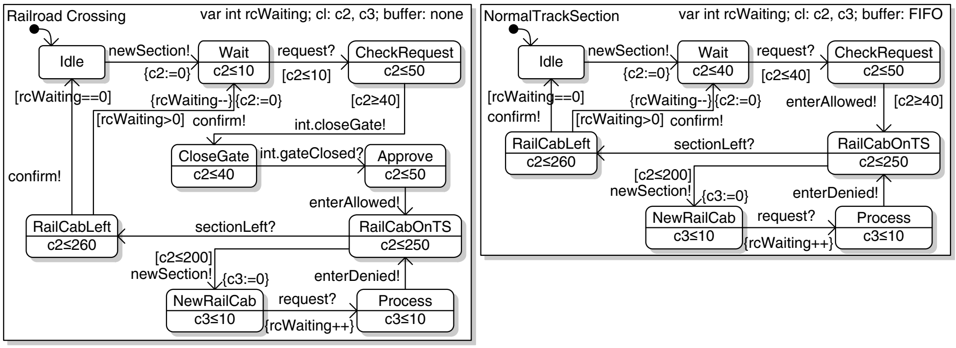

Figure 3: Refined Behavior for Railroad Crossings and Normal Track Sections

CheckRequest to RailCabOnTS is split into several transitions and intermediate locations that close the gates by using an internal message closeGate prefixed by int. After the gate responds that it is closed (gateClosed), the railroad crossing switches to Approve. Then, it sends enterAllowed and enters the RailCabOnTS location. In case of a normal track section, we only need to check whether the track is free. That, however, does not take as long as closing the gates at a railway crossing. Therefore, we may receive the input message later in a refined behavior utilizing the input buffer. Figure 3 shows the refined behavior for a normal track section. We relax the time guard of transition Wait to CheckRequest to $c2 \leq 40$. The remaining behavior remains unchanged.

After specifying the refined behavior models of Figure 3, we need to choose a suitable refinement definition for checking for a correct refinement. The choice of a suitable refinement definition depends on the verified properties to preserve as well as the characteristics of the (timed) automata used to model the system. Figure 4 summarizes the selection algorithm in form of a decision tree.

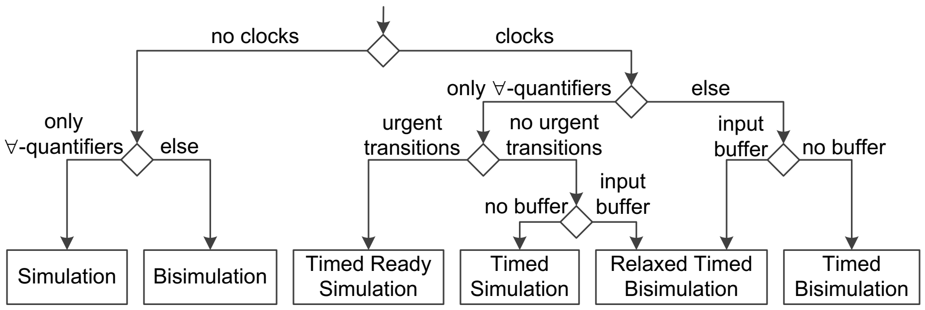

Figure 4: Decision Tree for Selecting a Refinement Definition

We can extract the necessary information for deriving a decision based on the decision tree by a syntactical analysis of both, the properties and the automata. For the first decision in the tree, we need to analyze whether the automata use clocks or not. Second, we check whether the properties only contain ∀-path quantifiers as, e.g., Property 1 in Section 2, or whether they also contain $\exists$-path quantifiers as, e.g., Property 2. Third, we analyze whether the abstract automaton uses urgent messages. Finally, we need the information whether the automata use a buffer for incoming messages provided by the developer.

In our example of Figure 3, the two refined automata need to preserve Properties 1 and 2 of Section 2. As a result, the decision algorithm selects the timed bisimulation for the refined model of the railroad crossing and it selects the relaxed timed bisimulation for the refined model of the normal track section. According to these definitions, the refinements given in Figure 3 are correct.

4 Test Automata Construction

Test automata have been introduced in [JLS00] as an approach for verifying refinements for timed automata. The basic idea of this approach is to encode the abstract model and the conditions for a correct refinement as a timed automaton T_A, called test automaton (Figure 5). These conditions define whether the developer is allowed to extend or restrict the time intervals for communication, or even to completely remove message sequences. Test constructs in T_A encode which changes are allowed and which are not, according to the conditions of the particular refinement definition (cf. Section 3).

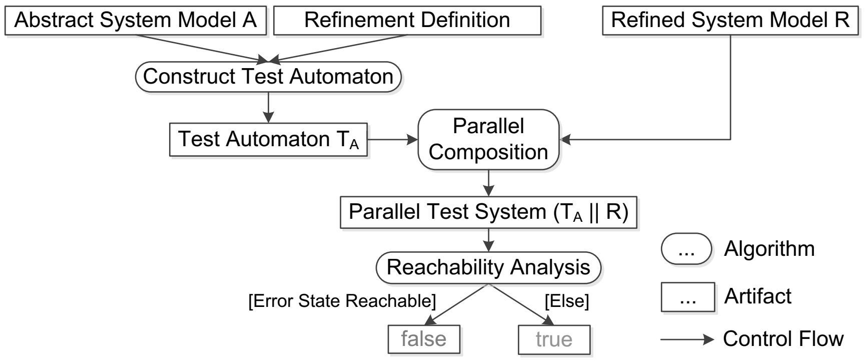

Figure 5: Verification using Test Automata

For the verification, we build the composed model $T_A \parallel R$, i.e. the parallel composition of T_A with the refined model R [BY03]. Then, we perform a reachability analysis on $T_A \parallel R$. During the reachability check, T_A communicates with the refined automaton R for detecting disallowed deviations from the message sequences of the abstract automaton A. If the conditions of the refinement definition are not fulfilled, the special error location Err in T_A becomes reachable. Otherwise, the refinement is correct.

Our test automaton construction generalizes and extends the construction as given in [JLS00]. The original construction only checks for a timed ready simulation. Our approach, on the contrary, supports checking all six refinement definitions given above (Figure 4). We extend the original approach of [JLS00] by introducing additional test constructs. This section explains how to construct T_A such that Err becomes reachable in $T_A \parallel R$ iff the selected refinement definition is *not* fulfilled for A and R.

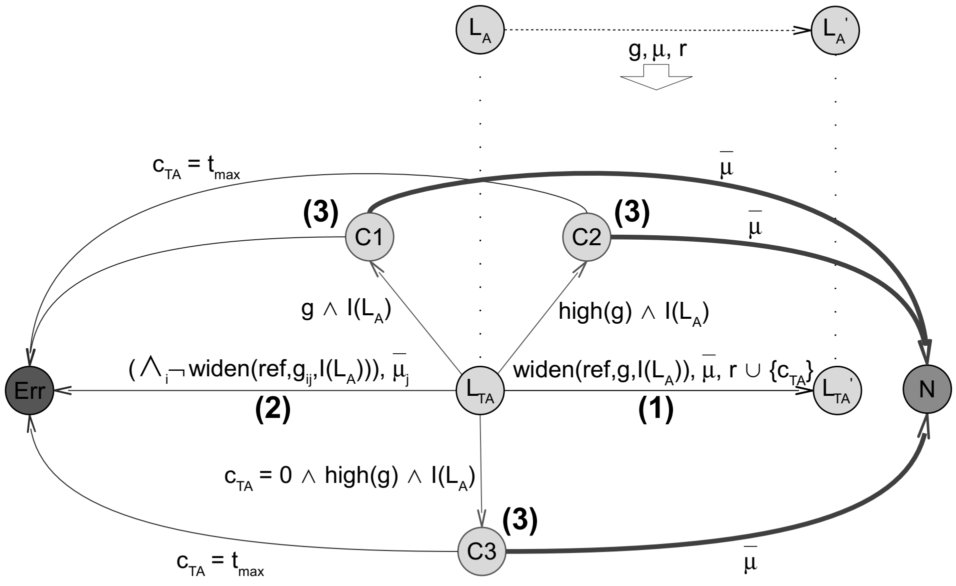

Figure 6: Construction Schema for our Test Automata

Figure 6 presents the schema for the construction of the part of T_A which is derived from a single transition $L_A \longrightarrow L_A{}'$ in the abstract model. T_A contains three kinds of test constructs, marked with (1)-(3) in Figure 6. These are explained in the following.

First, T_A must include all sequences of messages as defined by A, because all refinement definitions allow these sequences to be included in R. To model this in T_A, we define a corresponding transition $L_{TA} \longrightarrow L_{TA}{}'$ (1) for each transition $L_A \longrightarrow L_A{}'$ in A. The transition labels will be explained below.

Second, transitions $L_{TA} \longrightarrow Err$ (2) are defined for all sent and received messages which are not specified at outgoing transitions of L_A in A. One such transition is defined for each time interval in which a given message cannot be sent or received by A in L_A. If the developer refined A to R by adding communication not allowed by the refinement definition, these transitions make Err reachable. This so-called forbidden behavior must be checked for all refinement definitions, as none of them allows to add completely new message sequences in R (cf. Section 3).

Third, all variants of bisimulation (cf. Section 3) require that all sequences of messages specified in A are still included in R. Also, timed ready simulation requires all urgent communication in A to still be included in R. For checking this so-called required communication, T_A includes up to three locations $C1$, $C2$, and $C3$ (for each transition $L_A \longrightarrow L_A{}'$) (3). Transitions $L_{TA} \longrightarrow CX$, $CX \longrightarrow N$, and $CX \longrightarrow Err$ are established for each of these locations $CX(X \in [1,2,3])$. If the developer refined A to R by removing message sequences, T_A can reach Err via $C1$, $C2$, or $C3$. Successful tests for required communication lead to location N via $C1$, $C2$, or $C3$. Note that reaching N only indicates a successful test for one particular transition and does not allow any further conclusions about the correctness of the refinement.

The labels of the three test constructs for T_A depend on the specific refinement definition to check. We explain these labels in the following. We refer to [Bre10] for further technical details of the construction.

(1) Label definitions for allowed communication We create the labels for the transitions $L_{TA} \longrightarrow {L_{TA}}'$, modeling the message sequences allowed in R, as follows. Compared to the corresponding abstract transition $L_A \longrightarrow {L_A}'$, we invert the direction of all messages, i.e., input becomes output and vice versa. The symbol μ refers to the original message, $\overline{\mu}$ denotes the inverted one. This inversion ensures that T_A and R can synchronize in $T_A \parallel R$ whenever a sequence of messages is specified in both. Note that we use the $\parallel$-operator of UPPAAL[BY03].

Untimed simulation and bisimulation, as well as relaxed timed bisimulation each allow R to extend the time intervals defined in A. The time guard of each transition $L_{TA} \longrightarrow {L_{TA}}'$ in T_A is extended or removed accordingly by the function $widen$ (Figure 6). Depending on the refinement definition to check, given by ref in (Figure 6), $widen$ returns a modified time guard as follows. For untimed simulation and bisimulation, $widen$ returns a time guard that is always true, because time plays no role for these refinement definitions.

For relaxed timed bisimulation, the time guard returned by $widen$ depends on the transition $L_A \longrightarrow {L_A}'$. If $L_A \longrightarrow {L_A}'$ carries an input message, the returned time guard is always true. For these transitions, the relaxed timed bisimulation allows to extend time intervals arbitrarily. If $L_A \longrightarrow {L_A}'$ carries an output message, the returned time guard is the maximum of the upper bound of the time guard g of $L_A \longrightarrow {L_A}'$ and the invariants of L_A. This time guard models the condition of relaxed timed bisimulation that messages in R may not be sent later than in A. Earlier sending, however, is permitted by this time guard.

For the other three refinement definitions (timed simulation, timed bisimulation, and timed ready simulation), $widen$ returns the original time guard g, intersected with any invariants of L_A. This time guard defines the same time interval as specified for the abstract transition $L_A \longrightarrow {L_A}'$, because these refinements do not permit R to extend any time intervals of A.

(2) Label definitions for forbidden communication For each message μ_j in the alphabet of A which *is not sent or received* by any outgoing transition $L_A \longrightarrow L'_A$ in L_A, one transition $L_{TA} \longrightarrow Err$ is defined (cf. Figure 6). These transitions check for additional messages in R, which are not defined by A in the current location L_A. We need this check for all refinements, because none allows adding additional message sequences. The transition defined for a message μ_j carries the inverted message $\overline{\mu_j}$. This transition synchronizes with R, if R sends or receives the forbidden message μ_j. Then, Err becomes reachable in $T_A \parallel R$. We define the time guard of $L_{TA} \longrightarrow Err$ to be always true, because R may never offer μ_j, regardless of time.

For each message μ_j in the alphabet of A, which *is sent or received* by an outgoing transition $L_A \longrightarrow L'_A$ in L_A ($\mu = \mu_j$ in Figure 6), further transitions $L_{TA} \longrightarrow Err$ are defined. These transitions check whether the time intervals for sent or received messages in R are extended in comparison to the time intervals defined in A. We need this check for all timed refinements, because they do not allow R to extend the time intervals of A. Relaxed timed bisimulation allows extended time intervals in R, but forbids later sending of messages. To determine the time intervals where R may not define μ_j, we consider all transitions with μ_j in L_A. We write g_{ij} to refer to the time guard of the i-th transition with message μ_j in L_A. We create the conjunction of the negations of all these guards g_{ij}, each one modified by $widen$ (see above). The result are those time intervals in which μ_j is not defined in L_A. One transition $L_{TA} \longrightarrow Err$ is defined for each of these intervals. Each transition defined for μ_j carries the inverted message $\overline{\mu_j}$. If R sends or receives the forbidden message μ_j when no transition defining μ_j in L_A is enabled in A, R can synchronize with one of the transitions $L_{TA} \longrightarrow Err$ in T_A. Then, Err becomes reachable in $T_A \parallel R$.

(3) Label definitions for required communication $C1$ checks whether a message μ which is defined in A by a transition $L_A \longrightarrow {L_A}'$ is also defined by R during the time interval in which $L_A \longrightarrow {L_A}'$ is enabled. We need this check for timed bisimulation and timed ready simulation, because they do not allow R to restrict the time intervals for messages that were defined in A. For the transition $L_{TA} \longrightarrow C1$, we take over the time guard g of $L_A \longrightarrow {L_A}'$ and intersect it with the invariant $I(L_A)$ of the location L_A. This makes $C1$ reachable during the time interval in which $L_A \longrightarrow {L_A}'$ is enabled. The urgent transition $C1 \longrightarrow N$ carries the message $\overline{\mu}$. Whenever R can send or receive μ, it synchronizes with $C1 \longrightarrow N$, leading to N in $T_A \parallel R$. For $C1 \longrightarrow Err$ we define no time guard. Err becomes reachable via $C1 \longrightarrow Err$ in $T_A \parallel R$, whenever $C1 \longrightarrow N$ is not enabled. Because $C1 \longrightarrow N$ is urgent, it has precedence over $C1 \longrightarrow Err$ and prevents it from triggering while R sends or receives μ. If at any point in the time interval $g \wedge I(L_A)$, in which $L_A \longrightarrow {L_A}'$ is enabled, R *does not* send/receive μ, Err becomes reachable via $C1$ in $T_A \parallel R$.

$C2$ checks whether a message μ which is sent in A by a transition $L_A \longrightarrow {L_A}'$ is also sent by R at the end of the time interval in which $L_A \longrightarrow {L_A}'$ is enabled, or later. We need this check for the relaxed timed bisimulation. This refinement requires that the upper bounds of time intervals for sending messages in R are exactly the same as in A. The test construct (2) already checks that these time interval bounds are not raised in R. $C2$ checks that they are also not lowered, i.e. R must be able to send μ *at least* up to the upper bound in A. The latest time at which $L_A \longrightarrow {L_A}'$ is enabled is defined by the upper bound $high(g)$ of the time guard g and the invariant $I(L_A)$ of L_A, whichever is more restrictive. To ensure that $C2$ is entered only up to this time, we set the time guard of $L_{TA} \longrightarrow C2$ to $high(g) \wedge I(L_A)$. The urgent transition $C2 \longrightarrow N$ carries the message $\overline{\mu}$ to synchronize with R if it defines μ. We set the time guard of $C2 \longrightarrow Err$ to $c_{TA} = t_{max}$, to only allow reachability of Err after a maximum amount of time t_{max} has passed. The value

t_{max} is chosen high enough not to be reached in any actual execution of the system. If R, after reaching $C2$, still sends μ, the urgent transition $C2 \longrightarrow N$ forces T_A to enter N in $T_A \parallel R$. Reachability of Err is prevented in this case. If R never sends μ after this time, $c_{TA} = t_{max}$ eventually becomes true. Then, Err becomes reachable via $C2$ in $T_A \parallel R$.

$C3$ checks whether a message μ which is defined in A by a transition $L_A \longrightarrow L_A{}'$ is also defined by R at an arbitrary time. We need this check for untimed bisimulation and (the untimed condition of) relaxed timed bisimulation. These refinements require message sequences defined by A to also be defined by R but do not restrict timing. As above, the urgent transition $C3 \longrightarrow N$ carries the message $\overline{\mu}$ to synchronize with R when R defines μ. In the time guard of $L_{TA} \longrightarrow C3$, we check $c_{TA} = 0$ to make $C3$ reachable only right after T_A entered L_{TA}. The special clock c_{TA} is reset with every transition $L_{TA} \longrightarrow L_{TA}{}'$. Checking $c_{TA} = 0$ ensures that the time interval, in which R may fulfill the check by defining μ, starts directly after the last message exchange. It can not be reduced by T_A entering $C3$ at a later time. As for $C2$, we intersect the time guard of $L_{TA} \longrightarrow C3$ with $high(g) \wedge I(L_A)$. This intersection ensures that $C3$ is not reachable after the latest time the abstract transition $L_A \longrightarrow L_A{}'$, defining μ, is enabled. This is the case when L_A is entered with clock values already higher than the time guard of $L_A \longrightarrow L_A{}'$. Then, R is not required to define μ either and $C3$ is not reachable. We set the time guard of $C3 \longrightarrow Err$ to $c_{TA} = t_{max}$ (see above). Err only becomes reachable in $T_A \parallel R$ if R never defines the message μ after T_A entered location L_{TA} by a previous synchronization. Otherwise, the urgent transition $C3 \longrightarrow N$ synchronizes with R and forces T_A into N.

Implementation We implemented the automatic generation of the test automata based on a given refinement definition as described in [Bre10]. In addition, we support to verify that the refinement holds based on the parallel test system $T_A \parallel R$.

5 Related Work

We discuss related work from two areas. First, we review related work on approaches that support multiple refinement definitions. Second, we discuss related work on test automata.

Reeves and Streader [RS08a, RS08b] identify commonalities and differences of refinement definitions for process algebras and unify them in a generalized definition, but provide neither a selection nor a verification algorithm. Sylla et al. [SSdR05] present a refinement definition program including a refinement check where the refinement is parameterized by a particular LTL formula [BK08] such that only this particular formula is preserved. In contrast to our approach, both do not consider real-time properties. In [Bey01], Beyer introduces timed simulation for Cottbus Timed Automata which are a special kind of timed automata. We cover this refinement definition in our refinement check.

Test automata are used by [ABBL03] for model checking temporal properties specified in SBBL (Safety Model Property Language) on timed automata rather than verifying correct refinements. The test automata construction follows the same idea of encoding the conditions for correctness into the states of an automaton. The generated test constructs, however, are different. The approaches [GPVW96] and [Tri09] perform LTL model checking [BK08] on (timed) Büchi automata and encode the properties in automata as well. Again, the construction differs from our approach.

6 Conclusion and Future Work

In this paper, we present an automated automata-based refinement check for timed automata. Based on the (timed) automata used to specify the abstract and refined model of the system and the verified properties, we automatically select the most suitable refinement definition out of a set of six refinement definitions. The most suitable refinement definition is the least restrictive refinement definition that preserves all verified properties. Then, we automatically generate a so-called test automaton which encodes the abstract model and the conditions of the corresponding refinement. Our construction extends the construction of [JLS00] by additional test constructs. Using the test automaton, we verify whether all relevant properties still hold for the refined model.

Our approach enables developers of real-time systems to reuse (abstract) verified models of protocols that are specified in terms of (timed) automata. By verifying the correctness of refinements, we ensure that all verified properties are preserved. We relieve the developer from choosing a suitable refinement definition by automatically identifying the most suitable refinement based on the given model and the verified properties.

Future works will investigate whether we can relax the restrictions that currently apply to our test automaton construction. At present, the construction only allows to check for a correct refinement of a single timed automaton. Checking refinements for networks of timed automata requires to build a product automaton for the network [BY03]. We plan to investigate how the construction can be extended such that the explicit construction of the product automaton is not necessary. Furthermore, we want to extend the presented construction of refinements and test automata to dynamic communication structures. In a dynamic communication structure, the concrete communication topology may change during run-time which requires timed automata to be instantiated and deinstantiated dynamically.

Acknowledgments This work was developed in the course of the Collaborative Research Center 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft. Christian Brenner is supported by the International Graduate School Dynamic Intelligent Systems. We thank Matthias Becker, Steffen Becker, and Uwe Pohlmann for useful comments.

References

[ABBL03] L. Aceto, P. Bouyer, A. Burgueño, and K. Larsen. The power of reachability testing for timed automata. *Theor. Comput. Sci.*, 300(1-3):411–475, 2003.

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model-Checking in Dense Real-time. *Information and Computation*, 104:2–34, 1993.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[Bey01] D. Beyer. Efficient Reachability Analysis and Refinement Checking of Timed Automata Using BDDs. In T. Margaria and T. Melham, editors, *Proc. of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2001)*, volume 2144 of *Lecture Notes in Computer Science*, pages 86–91, 2001.

[BGK$^+$96] J. Bengtsson, D. Griffioen, K. Kristoffersen, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. In R. Alur and T. A. Henzinger, editors, *CAV 96*, number 1102 in LNCS, pages 244–256. Springer, July 1996.

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[Bre10] C. Brenner. Analyse von mechatronischen Systemen mittels Testautomaten. Masterarbeit, Universität Paderborn, August 2010.

[BY03] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003.

[GPVW96] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. of the 15th IFIP WG6.1 Intern. Sym. on Protocol Specification, Testing and Verification XV*, pages 3–18. Chapman & Hall, Ltd., 1996.

[HH11] C. Heinzemann and S. Henkler. Reusing Dynamic Communication Protocols in Self-Adaptive Embedded Component Architectures. In *Proc. of the 14th Intern. Sym. on Component Based Software Engineering*, CBSE '11, pages 109–118. ACM, June 2011.

[JLS00] H. E. Jensen, K. Larsen, and A. Skou. Scaling up Uppaal Automatic Verification of Real-Time Systems Using Compositionality and Abstraction. In *Proc. of the 6th Intern. Sym. on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT '00)*, pages 19–30. Springer, 2000.

[RS08a] S. Reeves and D. Streader. General Refinement, Part One: Interfaces, Determinism and Special Refinement. *Electron. Notes Theor. Comput. Sci.*, 214:277–307, June 2008.

[RS08b] S. Reeves and D. Streader. General Refinement, Part Two: Flexible Refinement. *Electron. Notes Theor. Comput. Sci.*, 214:309–329, June 2008.

[SSdR05] M. Sylla, F. Stomp, and W.-P. de Roever. Verifying parameterized refinement. In *Proc. 10th Intern. Conf. on Engineering of Complex Computer Systems (ICECCS 2005)*, pages 313 – 321, june 2005.

[SW07] W. Schäfer and H. Wehrheim. The Challenges of Building Advanced Mechatronic Systems. In *Future of Software Engineering (FOSE '07)*, pages 72–84. IEEE, 2007.

[Tri09] S. Tripakis. Checking timed Büchi automata emptiness on simulation graphs. *ACM Trans. Comput. Logic*, 10:15:1–15:19, April 2009.

[WL97] C. Weise and D. Lenzkes. Efficient Scaling-Invariant Checking of Timed Bisimulation. In *Proc. of STACS'97, LNCS 1200:pages 177–188*, pages 177–188. Springer, 1997.